REMARKS/ARGUMENTS

Claims 2-22 are pending in the present application. Claims 2-4, 10-12 and 22 were amended. Reconsideration of the claims is respectfully requested.

I. Objection to Drawings

The Examiner objected to the drawings, stating that Figures 1-3 should be designated by a legend such as --Prior Art--.

In response to this objection, the legend --PRIOR ART-- has been applied to each of the Figures 1-3.

II. 35 U.S.C. § 101

In the Final Office Action at page 2, item 5, the Examiner stated that Applicants' previous amendments to their Claims 2 and 12, set forth in their Response mailed July 16, 2007, are sufficient to overcome the prior 35 U.S.C. § 101 rejection of Claims 2-21. The Examiner further stated that the rejections are withdrawn. Applicants consider these statements to be correct, and consider further that a statement on page 4, item 9, which rejects Claims 2-6 and 12-16 under 35 U.S.C. § 101, is an oversight.

III. 35 U.S.C. § 101

The Examiner has rejected Claim 22 under 35 U.S.C. § 101. This rejection is respectfully traversed.

As Applicants stated in their prior Response, Claim 22 is considered to be statutory subject matter in view of provisions of **MPEP 2106**. Regarding non-statutory subject matter, the **MPEP** states:

> In this context, "functional descriptive material" consists of **data structures** and computer programs **which impart functionality when employed as a computer component**. (The definition of "data structure" is "a physical or logical relationship among data elements, designed to support specific data manipulation functions." The New IEEE Standard Dictionary of Electrical and Electronics Terms 308 (5th ed. 1993).) "Nonfunctional descriptive material" includes but is not limited to music, literary works and a compilation or mere arrangement of data.
>
> **When functional descriptive material is recorded on some computer-readable medium it becomes structurally and functionally interrelated to the medium and will be statutory in most cases since use of technology permits the function of the descriptive material to be realized.** Compare *In re Lowry*, 32 F.3d 1579, 1583-84, 32 USPQ2d 1031, 1035 (Fed. Cir. 1994) (claim to data structure stored on a computer readable medium that increases computer efficiency held statutory) and *Warmerdam*, 33 F.3d at 1360-61, 31 USPQ2d at 1759 (claim to computer having a specific data structure stored in memory held statutory product-by-process claim) with *Warmerdam*, 33 F.3d at 1361, 31 USPQ2d at 1760 (claim to a data structure *per se* held nonstatutory). (**emphasis added**) **MPEP 2106 (IV)(B)(1)**

Claim 22 recites clearly functional descriptive material, since such material imparts functionality when employed as a computer component. Moreover, the functional descriptive material of Claim 22 is recorded on "some" computer readable medium.

In the above context, the term "some" means "any" computer readable medium. The MPEP does not draw any distinctions between one type of media that is considered to be statutory and another type of media that is considered to be non-statutory. To the contrary, the MPEP clearly states that as long as the functional descriptive material is in "some" computer readable medium, it should be considered statutory. The only exception to this statement in the MPEP is functional descriptive material that does not generate a useful, concrete and tangible result, e.g., functional descriptive material composed completely of pure mathematical concepts that provide no practical result. Claim 22 clearly recites a useful, concrete and tangible result by using a plug-in class loader that is associated with and delegates to an identified class loader in order to load a plug-in class that is associated with a specified application class.

Accordingly, Claim 22 is directed to functional descriptive material that provides a useful, concrete and tangible result, and which is embodied on "some" computer readable medium. Therefore, Claim 22 is statutory, and the rejection thereof under 35 U.S.C. § 101 has been overcome.

IV. 35 U.S.C. § 103, Obviousness

The Examiner has rejected claims 2, 7-12 and 17-22 under 35 U.S.C. § 103, as being unpatentable over U.S. Publication No. 2004/0015936 to *Susarla et al.* (hereinafter "*Susarla*"). The Examiner has rejected claims 3-6 and 13-16 under 35 U.S.C. § 103 as being unpatentable over *Susarla* in view of teachings related to Figure 5B of the application, which Applicants provided to illustrate a problem of the prior. These rejections are respectfully traversed.

V. Teachings of Applicants

In their invention, Applicants provide a plug-in loader for each existing application class loader associated with a class loader hierarchy. In addition, a different plug-in class loader is provided for each class loader in the class loader hierarchy, wherein each plug-in class loader is associated with only a single class loader of the hierarchy, and each plug-in class delegates to its associated class loader. The class loader used to load a specified application class is identified, and the plug-in class loader associated with the identified class loader is then sued to load any plug-in class that is associated with the specified application class. Embodiments of the invention make it unnecessary to modify class path in order to

load plug-in classes, and also provide other benefits and advantages as described in the application, Moreover, embodiments of the invention provide a generic mechanism for any application to load plug-in classes at an appropriate level in the class loader hierarchy.

These teachings, set forth hereinafter, are found in the specification such as at page 18, lines 25 - page 19, line 2; page 19, line 24 - page 20, line 4; page 20, lines 11-20; and at Figure 6B.

> To solve the above mentioned problems, the present invention provides a set of plug-in class loaders in the class loader hierarchy. A plug-in class loader is provided for each class loader in the class loader hierarchy. Each plug-in class loader is associated with a single application class loader and is configured such that it delegates to its associated application class loader. **[page 18, line 25 – page 19, line 2]**
>
> As stated above, the present invention provides a plug-in class loader for each existing application class loader. Plug-in class loader **606** is provided for and delegates to application class loader 1 **602**. Plug-in class loader **608** is provided for and delegates to application class loader 2 **604**. Similarly, plug-in class loader **612** is provided for and delegates to system class loader **610**; plug-in class loader **622** is provided for and delegates to extension class loader **620**; and, plug-in class loader **632** is provided for and delegates to boot class loader **630**. **[page 19, line 24 – page 20, line 4]**
>
> **Figure 6B** illustrates the loading of plug-in classes by application classes using the loading class hierarchy of the present invention. The application class AppClass1 **652** is loaded by the boot class loader. Therefore, when the application class loads plug-in class Plug-In1 **654**, AppClass1 uses the plug-in class loader that delegates to the boot class loader to initiate the loading of Plug-In1. Similarly, since the application class AppClass2 **662** is loaded by the system class loader, AppClas2 uses the plug-in class loader that delegates to the system class loader to initiate the loading of the plug-in class, Plug-In2 **664**.**[page 20 lines 11-22]**

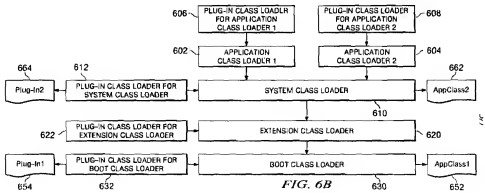

FIG. 6B

VI. Rejection of Claim 2

Applicants' Claim 2 now reads as follows:

2. A method for selecting a class loader to load a plug-in class within a class loader heirarchy, the method comprising the steps of:

generating a class loader hierarchy comprising a plurality of class loaders that includes two or more application class loaders, one for each of two or more application classes, wherein each of said application class loaders can selectively load its application

class, or delegate the loading of its application class to another class loader of said class loader hierarchy;

providing a different plug-in class loader for each class loader of said class loader hierarchy, wherein each plug-in class loader is associated with only a single class loader of said hierarchy, and each plug-in class loader delegates to its associated class loader;

identifying the class loader of said hierarchy that is used to load a specified one of said two or more application classes; and

using the plug-in class loader that is associated with and delegates to said identified class loader to load a given plug-in class that is associated with a said specified application class.

In rejecting Claim 2 as being obvious in view of *Susarla*, the Examiner stated the following:

> 13. **Regarding Claims 2, 12 and 22:** Susarla discloses a method for selecting a class loader for a plug-in, the method comprising:
> generating a class loader hierarchy (Fig. 4) comprising a plurality of class loaders that includes two or more application class loaders, one for each of two or more application classes (par. [0101] "Each application may include an application class loader 202"), wherein each of said application class loaders can selectively load its application class, or delegate the loading of its application class to another class loader of said class loader hierarchy (par. [0017] "a delegation mechanism");
> providing a plug-in class loader for each application class loader in the class loader hierarchy (Fig 4, Module Class Loader 204A), wherein each plug-in class loader delegates to its corresponding class loader (Fig. 4 Application Class Loader 202)
> identifying the class loader of said hierarchy that is used to load a specified one of said two or more application classes (par. [0138] "The class loader controller may ... locate the appropriate class loader in the stack"); and
> using the plug-in class loader that is provided for and delegates to said identified class loader to load a plug-in class that is associated with said specified application class (par. [0138] "The class loader controller may . .. invoke the located class loader").
>
> 14. Susarla does not explicitly disclose providing a different plug-in class loader for each class loader in the hierarchy (specifically, the System Class Loader 200 of Fig. 4 is not show as having a plug-in class loader).
>
> 15. It would have been obvious to one of ordinary skill in the art at the time the invention was made to provide a different plug-in class loader (Fig. 4, Module Class Loader 204A) for each class loader in the Hierarchy (i.e. Fig. 4, System Class Loader 200) in order to provide 'dynamic class reloading' (see Abstract) at the system level as opposed to the application level (par. [0007] "The system class loader loads the standard classes and the application server core classes, and the application class loader loads the user-defined classes"). **[Final Office Action dated 09/24/2007, pps. 5-6]**

Pertinent teachings of Susarla are found at the abstract, paragraphs [0056], [0059], [0101] and [0138] and at Figure 4 thereof. These sections are as follows:

> A system and method for providing dynamic class reloading using a modular, pluggable and maintainable class loader is described. Each application in an application server (or alternatively in any implementation) may include a dynamic class loader module. The class loader module may include a hierarchical stack of class loaders. Each module in the

application may be associated with its own class loader. Each class loader may be responsible for loading one or more classes. When a class is changed, the changed class may be detected by the class loader module. Any notification for a class change may come to the class loader controller so that the concerned class loader can be replaced. The class loaders for all classes that depend on the changed class may also be replaced. The replaced class loaders may then reload the affected classes. **(abstract)**

[0056] A system and method for providing dynamic class reloading using a modular, pluggable and easily maintainable class loader framework is described. In one embodiment, the dynamic class reloading mechanism as described herein may be applied to Java.TM. applications. Other embodiments may be applied to applications written in other programming languages. Each application in an application server (or alternatively in any implementation) may include a dynamic class loader module. The class loader module may include a hierarchical stack of class loaders. Each class loader may have one parent class loader and zero or more child class loaders. Each module in the application may be associated with its own class loader; in other words, there may be one class loader for each module. Each class loader may be responsible for loading one or more classes.

[0059] At some point, one or more of the classes used by the application may be changed. For example, a programmer may make a modification to a class. The application may detect that a class has been changed. In one embodiment, the application may include a dirty class monitor that may monitor classes used by the application and detect when any of the classes have been changed. The class loader for the class may be replaced with a new version of the class loader configured to load the changed class. In one embodiment, the dirty class monitor may notify the class loader controller that the class has been changed. The class loader controller may then locate the class loader responsible for loading the class in the hierarchical stack of class loaders. The class loader controller may then replace the class loader with the new class loader. If there are one or more classes that depend on the class to be reloaded, the class loaders responsible for reloading the dependent classes may be located and replaced as well. If one or more of the dependent classes are loaded by the same class loader that is responsible for loading the changed class, then the class loader may only be replaced once. After replacing the class loader(s), the new class loader may load the changed class (which may be referred to as "reloading the class"). In one embodiment, dependent classes, if any, may also be reloaded by their respective class loaders. In one embodiment, the class loader controller may invoke each of the necessary class loaders to reload the class(es) that need to be reloaded in response to the change in the class.

[0101] FIG. 4 illustrates a class loader stack for an application according to one embodiment. Each application may include an application class loader **202** that may loads cross-module classes, utility classes, interfaces, stubs and skeletons for the application. A system class loader **200** is shown as the parent class loader of the application class loader **202**. At the layer below the application class loader **202** are the module level loaders **204**. In one embodiment, below the module level loaders are Web class loaders **206** and EJB.TM. class loaders **208**.

[0138] FIG. 7 is a flowchart illustrating a method for providing dynamic class reloading in applications according to one embodiment. As illustrated at **300**, one or more class loaders may, when necessary, load classes for an application. In one embodiment, the application may include a class loader module that may include a hierarchical stack of class loaders that are each configured to load one or more classes for the application when invoked. In one embodiment, a class loader controller may provide an interface to the stack of class loaders that is configured for use in invoking the class loaders to load the classes. The class loader controller may be configured to receive a request to load a class,

and may, in response to receiving the request, may first locate the appropriate class loader in the stack of class loaders and then invoke the located class loader.

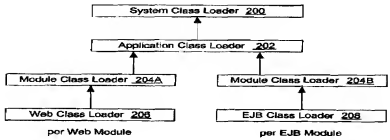

Fig. 4

In order to establish a *prima facie* case of obviousness under 35 U.S.C. § 103, *Graham v. John Deere Co. of Kansas City*, 383 U.S. 1 (1966) requires determining, respectively, the scope and content of the prior art, the differences between the prior art and the claims at issue, and the level of ordinary skill in the pertinent art. Also, the prior art or combined references much teach or suggest all of the claim limitations. The suggestion to make the claim combination must be found in the prior art, not in the Applicants' disclosure. *In re Vaek*, 20 U.S.P.Q.2d 1438 (Fed. Cir. 1991). Moreover, in accordance with **MPEP § 2142.02**, each prior art reference must be considered in its entirety, i.e., as a whole, including portions that would lead away from the claimed invention. *W.L. Gore & Associates, Inc. v. Garlock, Inc.*, 220 U.S.P.Q. 303 (Fed. Cir. 1993). A third essential requirement for establishing a *prima facie* case, set forth in **MPEP § 2143.01**, is that the proposed modification cannot render the prior art unsatisfactory for its intended purpose.

In the present case, not all of the features of the claimed invention have been properly considered, and the teachings of the reference itself do not teach or suggest the claimed subject matter to a person of ordinary skill in the art. For example, *Susarla* fails to teach or suggest, in the over-all combination of Claim 2, either of the following features:

(1) Providing a different plug-in class loader for each class loader of the class loader hierarchy, wherein each plug-in class loader is associated with only a single class loader of the hierarchy, and each plug-in class loader delegates to its associated class loader (hereinafter "Feature (1)").

(2) Using the plug-in class loader that is associated with and delegates to the identified class loader to load a plug-in class that is associated with the specified application class (hereinafter "Feature (2)").

VII. Claim 2 Distinguishes over Cited Reference

Feature (1) of Applicants' Claim 2 is directed to providing a different plug-in class loader for each class loader in a class loader hierarchy. Moreover, each plug-in class loader is associated with only a single class loader of the hierarchy, and each class loader delegates to its corresponding class loader. This feature is disclosed in Applicants' specification, such as at page 20, lines 11-22, taken together with Figure 6B of Applicants' drawings, and also at page 24, lines 3-9. Figure 6B shows a class loader hierarchy comprising the class loaders 610, 620 and 630, and further comprising class loader 602 and 604 for application classes 1 and 2, respectively. Figure 6B further shows each of the plug-in loaders 606, 608, 612, 622 and 632, wherein each class loader is provided with a different plug-in class loader, and each plug-in class loader is associated with only one of the class loaders 602-604 or 610-630. As taught in the specification at page 24, lines 6-9, Feature (1) of Claim 2 is essential in providing the benefit of a generic mechanism, for enabling any application to load plug-in classes at an appropriate level in the class loader hierarchy.

In the Final Office Action, the Examiner acknowledged that *Susarla* fails to show the Feature (1) recitation of providing a different plug-in class loader for each class loader in the hierarchy. *Susarla*, in fact, appears to provide no teaching whatever in regard to the loading of plug-ins, which are extensions meant to extend the behavior of application classes, as is known by those of skill in the art. To the contrary, *Susarla* is concerned with an entirely different issue, that is, the dynamic reloading of classes, which may become necessary when classes used by an application have been changed. As taught by *Susarla* at paragraph [0059], classes can change, for example, when a programmer makes a modification to a class. In the abstract and at paragraph [0056], *Susarla* teaches that each application in a server may include a dynamic class loader module, wherein the module comprises a hierarchical stack of class loaders. Each class loader is configured to load one or more classes for the application when invoked. As taught at paragraph [0101], Figure 4 shows a hierarchical stack of this type.

In the Final Office Action, the Examiner stated that it was obvious to provide a different plug-in class loader for each class loader in a hierarchy. However, only the abstract, paragraph [0007] and Figure 4 of *Susarla* were cited to support this contention. As discussed above, the abstract generally discloses providing dynamic class reloading, with a module that includes a hierarchical stack of class loaders. Paragraph [0007] discusses a system class loader as a parent of class loaders, and also discusses deficiencies of the prior art in regard to reloading classes for

a class change. Figure 4 depicts a system class loader 200 as the parent class loader of an application class loader 202.

It is readily apparent that each of these cited sections of *Susarla*, including Figure 4, fails to disclose or suggest, or provide any reason for, the Feature (1) recitation of providing a different plug-in class loader for each class loader in a class loader hierarchy, wherein each plug-in class loader delegates to its associated class loader. Moreover, these sections of *Susarla* neither disclose nor suggest any structure or elements that are equivalent to the plug-in loader configuration of Applicants' Feature (1). The cited sections of *Susarla* also fail to disclose or suggest the Feature (1) recitation that each plug-in class loader is associated with only a single class loader of the hierarchy. The selections likewise fail to provide any reason as to why such limitation would be beneficial in the arrangement of *Susarla*.

The above deficiencies of *Susarla*, in regard to Feature (1) of Applicants' Claim 2, are not found or overcome elsewhere in *Susarla*. Nowhere does Susarla disclose plug-in class loaders or equivalent elements, in accordance with the recitation of Feature (1). Neither does *Susarla* teach or provide any reason to make the changes needed to reach Feature (1), in the combination of Claim 2. As described above, the objectives and purposes of *Susarla* are quite different from those of Applicants. Accordingly, Feature (1) of Claim 2 is clearly non-obvious in view of *Susarla*.

Feature (2) of Claim 2 recites using the plug-in class loader that is associated with and delegates to the identified class loader to load the plug-in class that is associated with the specified application class. *Susarla*, of course, cannot show this feature, since it fails to disclose the plug-in class loader required for Feature (2), as was discussed above in connection with Feature (1).

Moreover, paragraph [0138] of *Susarla*, cited in the Final Office Action in connection with Feature (2), once again discloses a class loader module that includes a hierarchical stack of class loaders, for dynamic class reloading as discussed above. Paragraph [0138] further teaches that a class loader controller can be configured to receive a request to load a class, and in response to receiving a request, may locate the appropriate class loader in the stack, and then invoke it. Clearly, this teaching of *Susarla* fails to show or suggest using a plug-in that is associated with and delegates to an identified class loader, in order to load a plug-in class, as recited by Feature (2) of Claim 2.

VIII. Claim 3 Distinguishes over Cited Reference

Claim 3 depends from Claim 2, and is considered to distinguish over the art for at least the same reasons given in support thereof. In addition, Claim 3 distinguishes over the art in reciting the

feature that an application file is associated with the plug-in classes, and the plug-in class loader that is associated with the identified class loader is enabled to locate the given plug-in class by specifying a single configuration value in the application file. This feature is taught in Applicants' specification, such as at page 20, lines 24-28, but is neither disclosed nor suggested by *Susarla.*

IX. Claims 10 and 11 Distinguish over Cited Reference

Claim 10 depends from Claim 2, and is considered to distinguish over the prior art for at least the same reasons given in support thereof. In addition, Claim 10 distinguishes over the art in reciting the feature that a particular one of the application classes is loaded by a particular class loader of the class loader hierarchy, and first and second plug-in classes associated with respective first and second application classes respectively are each specified to use the class loader of the particular application, and the first and second plug-in classes are both loaded by the plug-in class loader that is associated with and delegates to the particular class loader. This feature is recited in Applicants' specification, such as at page 21, line 23 – page 22, line 7. *Susarla* however, neither shows nor suggests this feature.

Claim 11 depends from Claim 10, and is considered to distinguish over the art for at least the same reasons given in support thereof. In addition, Claim 11 is considered to distinguish over the art in reciting the feature that the first plug-in class and the second plug-in class are able to share data. *Susarla* neither shows nor suggests this feature.

X. Remaining Claims Distinguish over the Cited Reference

Independent Claims 12 and 22 respectively incorporate subject matter similar to the patentable subject matter of Claim 2, and are each considered to distinguish over the art for at least the same reasons given in support thereof.

Claims 4-9 and 13-21 depend from independent Claims 2 and 12, respectively, and are each considered to distinguish over the art for at least the same reasons given in support thereof.

XI. Conclusion

It is respectfully urged that the subject application is patentable over the *Susarla* reference and is now in condition for allowance.

The Examiner is invited to call the undersigned at the below-listed telephone number if in the opinion of the Examiner such a telephone conference would expedite or aid the prosecution and examination of this application.

DATE: December 26, 2007

Respectfully submitted,

/James O. Skarsten/

James O. Skarsten
Reg. No. 28,346
Yee & Associates, P.C.
P.O. Box 802333
Dallas, TX 75380
(972) 385-8777
Attorney for Applicants